Facilitating the Portability of User Applications in Grid Environments

Paul Z. Kolano

NASA Advanced Supercomputing Division, NASA Ames Research Center,
M/S 258-6, Moffett Field, CA 94035 U.S.A.
kolano@nas.nasa.gov

Abstract. Grid computing promises the ability to connect geographically and organizationally distributed resources to increase effective computational power, resource utilization, and resource accessibility. For grid computing to be successful, however, users must be able to easily execute the same application on different resources. Different resources, however, may be administered by different organizations with different software installed, different file system structures, and different default environment settings. Even within the same organization, the set of software installed on a given resource is in constant flux with additions, upgrades, and removals. Users cannot be expected to understand all of the idiosyncrasies of each resource they may wish to execute jobs on, thus must be provided with automated assistance. This paper describes a new OGSI-compliant grid service (the *Portability Manager*) that has been implemented as part of NASA's Information Power Grid (IPG) project to automatically establish the execution environment for user applications.

1 Introduction

Grid computing [6] promises the ability to connect geographically and organizationally distributed resources to increase effective computational power, resource utilization, and resource accessibility. Real world experiences with grids [1,11], however, have had mixed results. While gains in computational power were eventually achieved, they were only made a reality after significant efforts to get user applications running on each suitable resource. Differences across resources in installed software, file system structures, and default environment settings required manually transferring dependent software and setting environment variables. This problem is common even in non-grid environments. Users frequently encounter missing or incompatible shared libraries (.so files) on Unix systems or missing dynamic link libraries (.DLL files) on Windows systems when attempting to execute binaries that have been transferred from a similar system. Even in a language that is designed for portability, such as Java, this same problem exists. That is, a Java application can only be executed on a system that has all of the classes installed on which it depends. If all dependent software is present, an application still may not be able to execute if the environment variables are not set such that it can find that software. In grid environments, this problem is to allow different versions of an executable to be staged to a machine based on its processor architecture type and operating system version. Executables are retrieved from a network-based executable repository. This system only supports executables, however, and has no support for shared libraries, Java classes, or Perl or Python programs, nor does it support automated dependency analysis.

The Uniform Interface to Computing Resources (UNICORE) [4] allows jobs to be built from platform-independent abstract job operations, which are translated into concrete operations that can be executed on an actual system. The translation relies on a static configuration file located on each resource describing the software installed there. For example, an abstract job executing "ls" would be mapped using the configuration file to a concrete job executing "/bin/ls". This approach requires extra administration every time software is added to, removed from, or updated on a system, it only supports executables, and it only allows software to run on systems that already have all required software installed.

The Automatic Configuration Service of [10] was implemented to automatically manage the installation and removal of software for component-based applications according to user-specified dependency information. This service has goals similar to those of the Portability Manager, but is implemented as a CORBA service as opposed to an OGSI-compliant service. A limitation of this service is that the user must fully specify all dependencies manually. There is also no discussion of managing environment variables, which are required for an application to find installed software and which differ according to software type. In addition, this service uses a centralized repository, thus cannot take advantage of software individually deployed by users.

Installers, package managers, and application management systems [3] are typically used to manage the software installed on standalone systems and systems on the same network. While these approaches greatly increase the ability of system administrators to provide a consistent and stable set of software across an organization's resources, they are only of use when the administrator knows what software will be needed. Since grids enable users from different organizations with different software requirements to share resources, these mechanisms do not provide the necessary level of support.

Replica management systems such as Reptor [8] provide high-level mechanisms for managing the replication, selection, consistency, and security of data to provide users with transparent access to geographically distributed data sets. Much of this functionality is also suitable for managing software across grid resources and is, in fact, the basis of part of the Portability Manager. Replica management systems do not address software specific issues, however, such as automatic dependency analysis and environment variable settings.

3 NASA Information Power Grid

NASA's Information Power Grid (IPG) [9] is a computational and data grid spanning a number of NASA centers that consists of various high performance supercomputers, storage systems, and data gathering instruments scattered across

1. Determine the software that the execution operation application requires
2. Provide a location for that software on the execution operation host by:
    (a) Determining if the software exists on the execution operation host
    (b) Finding a source for any missing software
    (c) Copying missing software to the execution operation host
3. Set environment variables based on provided software locations

A list of dependencies is associated with each execution operation. A dependency consists of basic requirements including a type, a name, a version range, and a feature list as well as information gathered during processing including a source host and path, a target path, and an "analyzed flag" to indicate its analysis status. The Portability Manager currently supports five software types: executables, shared libraries, Java classes, and Perl and Python programs. The dependency name contains a canonical name for the software depending on its type (e.g. ls, libc, java.util.List, File::Basename, xml.sax.xmlreader, etc.). The version range consists of a minimum and/or maximum version required. The feature list contains features that the dependency must support. For example, the application might require the w3m browser compiled with SSL support. Currently, versions are only supported for shared libraries and features are not yet supported as a consistent way to determine these automatically has not yet been determined.

The source host and path, target path, and analyzed flag are used to store information as processing proceeds. Stages are only executed if the information they provide has not already been gathered. Thus, a job for which the execution environment has already been fully established can be sent through the Portability Manager without effect. This allows the user to have complete control of job processing. A user can execute stages individually, can specify dependencies manually, can turn analysis off, can specify an exact source for software, can specify a location where software already exists, or any combination thereof. The Portability Manager will fill in any gaps in the environment left by the user or return the job unchanged if no modifications are necessary.

Although the Portability Manager makes its best attempt to establish the execution environment for a job, it is not possible to guarantee that the resulting environment will always be suitable. There are three scenarios for which such a guarantee cannot be made:

1. Required software A does not exist or cannot be located on the target system and no source for A can be found
2. Required software A depends on software B, which depends on software C, but the file for B cannot be located for analysis
3. Required software A depends on software B, but the analysis techniques used on A are inadequate to determine B is a dependency

Since executing a job for which the execution environment has not been fully established leads to wasted CPU cycles, it is desirable to notify the user prior to job execution. For the third scenario, nothing can be done besides documenting the limitations of the analysis techniques. The existence of the first two scenarios,

```
# addall.py (add #'s from stdin)
import sys
import string
import Adder
adder = Adder.Adder()
for line in sys.stdin.readlines():
  n = string.atoi(line)
  adder.add(n)
print adder.sum()

# Adder.py (maintain sum)
class Adder:
  def __init__(self):
    self.value = 0
  def add(self, n):
    self.value += n
  def sum(self):
    return self.value
```

Fig. 1. Example job

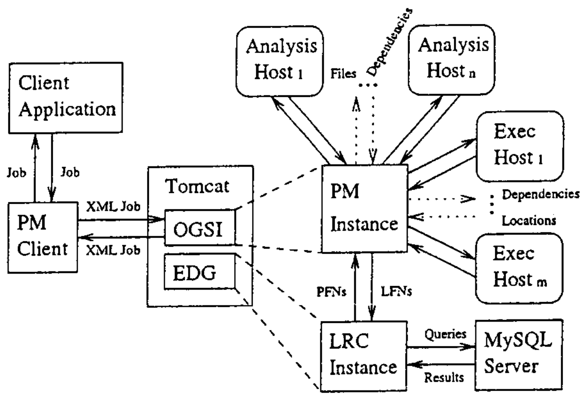
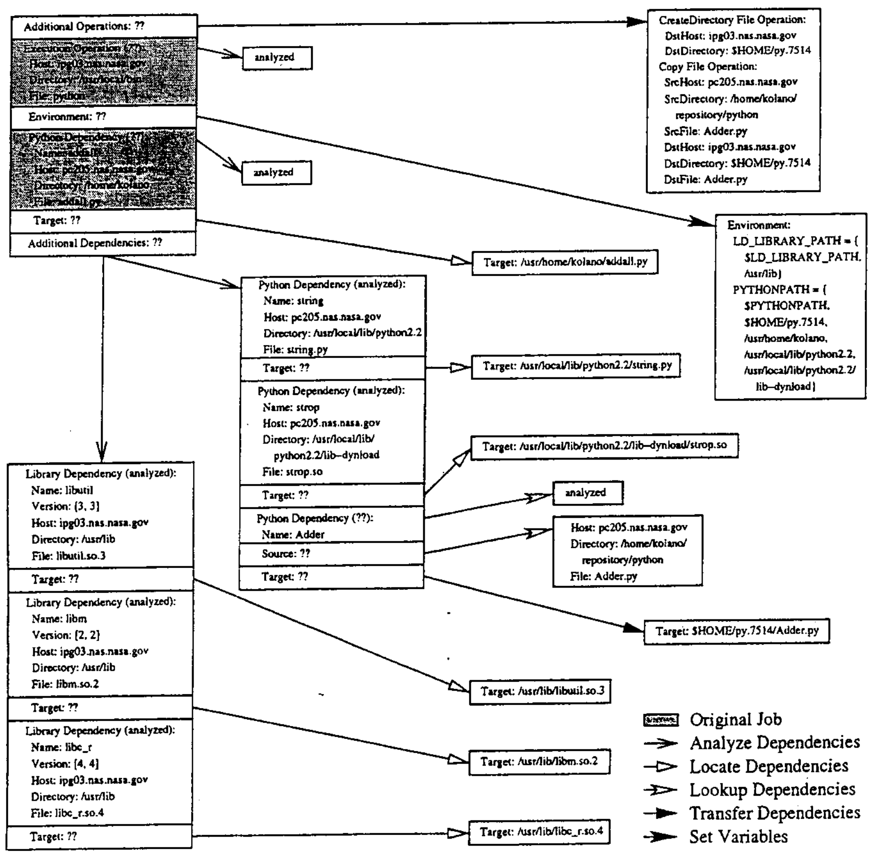


Fig. 2. Portability Manager implementation



Additional Operations: ??
Environment: ??
Target: ??
Additional Dependencies: ??
analyzed
analyzed
CreateDirectory File Operation:
DstHost: ipg03.nas.nasa.gov
DstDirectory: $HOME/py.7514
Copy File Operation:
SrcHost: pc205.nas.nasa.gov
SrcDirectory: /home/kolano/repository/python
SrcFile: Adder.py
DstHost: ipg03.nas.nasa.gov
DstDirectory: $HOME/py.7514
DstFile: Adder.py
Environment:
LD_LIBRARY_PATH = { $LD_LIBRARY_PATH, /usr/lib }
PYTHONPATH = { $PYTHONPATH, $HOME/py.7514, /usr/home/kolano, /usr/local/lib/python2.2, /usr/local/lib/python2.2/lib-dynload }
Target: /usr/home/kolano/addall.py
Python Dependency (analyzed):
Name: string
Host: pc205.nas.nasa.gov
Directory: /usr/local/lib/python2.2
File: string.py
Target: ??
Target: /usr/local/lib/python2.2/string.py
Python Dependency (analyzed):
Name: strop
Host: pc205.nas.nasa.gov
Directory: /usr/local/lib/python2.2/lib-dynload
File: strop.so
Target: ??
Target: /usr/local/lib/python2.2/lib-dynload/strop.so
analyzed
Python Dependency (??):
Name: Adder
Source: ??
Target: ??
Host: pc205.nas.nasa.gov
Directory: /home/kolano/repository/python
File: Adder.py
Target: $HOME/py.7514/Adder.py
Library Dependency (analyzed):
Name: libutil
Version: {3, 3}
Host: ipg03.nas.nasa.gov
Directory: /usr/lib
File: libutil.so.3
Target: ??
Target: /usr/lib/libutil.so.3
Library Dependency (analyzed):
Name: libm
Version: {2, 2}
Host: ipg03.nas.nasa.gov
Directory: /usr/lib
File: libm.so.2
Target: ??
Target: /usr/lib/libm.so.2
Library Dependency (analyzed):
Name: libc_r
Version: {4, 4}
Host: ipg03.nas.nasa.gov
Directory: /usr/lib
File: libc_r.so.4
Target: ??
Target: /usr/lib/libc_r.so.4
Original Job
Analyze Dependencies
Locate Dependencies
Lookup Dependencies
Transfer Dependencies
Set Variables

Fig. 3. Stages of job transformation

including bash, csh, ksh, sh, tcsh, and zsh. A variable "var" can be read from a shell "<shell>" using "<shell> -c 'echo $var"'. Variables gathered include PATH, LD_LIBRARY_PATH and variants, CLASSPATH, JAVA_HOME, PERLLIB and variants, and PYTHONPATH. Once the paths are set, files are located by type, using "ls" for executables and libraries and the corresponding interpreter for Java, Perl, and Python dependencies. It is assumed that if the interpreter is not available, then no dependencies of that type exist on the system.

After this stage, Figure 3 shows that all dependencies of the example job have been located on the target system except for the Adder Python dependency.

4.3 Dependency Lookup

Ideally, after the analysis and location stages, every dependency has either been located on the target system or a source for it has been found during analysis. Since there is no guarantee that this will be the case, however, a final attempt is made to find any unresolved dependencies in a software catalog. This catalog is based on the European DataGrid (EDG) Local Replica Catalog (LRC), which is part of a complete replica management system [8]. LRCs map logical file names (LFNs) to physical file names (PFNs) and are normally used for storing the locations of data sets. In this case, an LRC is used to map LFNs constructed from a software's type, name, supported operating system, and version to a location on some system where that particular software resides. In addition, each PFN may have a set of attributes associated with it. Since dependency analysis has already been performed by this stage, the Portability Manager uses these attributes to store the LFNs of pre-identified dependencies associated with each PFN, which are recursively added as dependencies and looked up as necessary.

Using a catalog instead of a repository allows for a flexible approach to software management. As long as a resource is accessible to the file transfer mechanisms of the IPG Job Manager, the software on that resource can be utilized by the Portability Manager. If an organization wishes to have a permanent repository, they can dedicate a set of resources to the task with an appropriate repertoire of software and map LFNs into the file systems of those resources. Otherwise, the LFNs can simply point to the locations of software on existing systems. The design also allows users to manage personal software repositories. The Portability Manager provides a user interface to add and remove mappings from LFNs in a personal namespace based on their grid identity to the PFNs of choice. Thus, users can maintain a collection of software that they frequently use on their personally selected resources, which will be utilized by the Portability Manager as a source for the software required by their jobs. For a given LFN, the current implementation first selects the user's PFN, if it exists, or if not, selects the first matching PFN from the main catalog. Future versions of the Portability Manager will perform more intelligent selection based on locality, reliability, etc.

After this stage, Figure 3 shows that the one dependency without a source or target location, the Adder Python dependency, now has a source. In addition, it has been marked as "analyzed" based on its dependency information in the software catalog, which indicated that no additional software was required.

framework [7]. In the OGSA model, all grid functionality is provided by named *grid services* that are created dynamically upon request. The newest version of Globus, version 3.0 (GT3), is the reference implementation of OGSI and provides the functionality of GT2 as grid services.

Figure 2 shows the current implementation of the Portability Manager. In this figure, a client application uses the Portability Manager client API to request the establishment of the execution environment for a given job. The Portability Manager client converts the Java job object into XML for transmission to an Apache Tomcat server running an OGSI container. The OGSI container creates an instance of the Portability Manager and invokes its "establishEnvironment" method with the given job. The Portability Manager uses the OGSI GRAM service to execute the analysis script on each host with files requiring analysis in parallel. All jobs are executed using the grid credentials of the client application user, thus users are not given any additional privileges beyond what they normally have. After all dependencies have been gathered, the location script is then executed in parallel on each execution operation host with unresolved dependencies. For any dependencies that could not be located or for which no source could be found, an instance of the EDG LRC is queried in an attempt to find a source. At this point, the Portability Manager sets up the return job to copy dependencies as necessary and sets the environment variables appropriately. The job is returned in XML to the Portability Manager client, which converts the job back into a Java object for the client application.

The Portability Manager has been fully tested on FreeBSD systems and the analysis and location scripts have been tested on IRIX, SunOS, and FreeBSD. It has not yet been deployed in the NASA IPG due to the limitations of the current release of the GT3 toolkit. Currently, GT3 is only available in alpha form, which is not yet suitable for production IPG usage and does not support IRIX systems, which make up the bulk of the IPG. For the same reason, the Portability Manager has not yet been integrated with the IPG Resource Broker or Job Manager, but exists as a standalone service. The current Resource Broker and Job Manager are built on top of GT2. The Portability Manager will be integrated with the next versions of these services, which will be OGSI-compliant.

6 Conclusions and Future Work

This paper has described the IPG Portability Manager, which is an OGSI-compliant grid service that has been implemented to automatically establish the execution environment for user applications. The Portability Manager analyzes applications to determine their software dependencies, locates the software on the target system, if possible, or elsewhere, if not, arranges the transfer of software as necessary, and sets the environment variables to allow each application to find its required software. The Portability Manager has a flexible design that gives the user considerable control over job processing including choosing which steps to perform and managing the source for frequently used software in a personal software catalog. The Portability Manager allows users to execute